John von Neumann Institute for Computing

Compiler Support for Efficient Instrumentation

Oscar Hernandez, Haoqiang Jin, Barbara Chapman

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 661-668, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Compiler Support for Efficient Instrumentation

Oscar Hernandez[1], **Haoqiang Jin**[2], **and Barbara Chapman**[1]

[1] Computer Science Department
University of Houston
501 Phillip G. Hoffman, Houston, Texas
E-mail: {oscar, chapman}@cs.uh.edu

[2] NASA Advanced Supercomputing Division, M/S 258-1
NASA Ames Research Center, Moffet Field, CA
E-mail: hjin@nas.nasa.gov

We are developing an integrated environment for application tuning that combines robust, existing, open source software - the OpenUH compiler and performance tools. The goal of this effort is to increase user productivity by providing an automated, scalable performance measurement and optimization system. The software and interfaces that we have created has enabled us to accomplish a scalable strategy for performance analysis, which is essential if performance tuning tools are to address the needs of emerging very large scale systems. We have discovered that one of the benefits of using compiler technology in this context is that it can direct the performance tools to decide which regions of code they should measure, selectively considering both coarse grain and fine grain regions (control flow level) of the code. Using a cost model embedded in the compiler's interprocedural analyzer, we can statically assess the importance of a region via an estimated cost vector that takes its size and the frequency with which it is invoked into account. This approach enables us to set different thresholds that determine whether or not a given region should be instrumented. Our strategy has been shown to significantly reduce overheads for both profiling and tracing to acceptable levels. In this paper, we show how the compiler helps identify performance problems by illustrating its use with the NAS parallel benchmarks and a cloud resolving model code.

1 Introduction

Tracing and profiling play a significant role in supporting attempts to understand the behaviour of an application at different levels of detail and abstraction. Both approaches can be used to measure a variety of different kinds of events, where memory profiling/tracing are the most expensive. Profiling aggregates the results of event tracking for phases in the code or for the entire execution of the program, while tracing can keep track of every single instance of an event at run time, and captures patterns over time. Tools such as TAU[1] support both profiling and tracing. KOJAK[2], on the other hand, uses tracing to find performance patterns. This can enable the tools to detect communication and synchronizations problems for both MPI and OpenMP codes, as well as giving support for hardware counter patterns.

When applications are deployed on systems with thousand of processors, it is critical to provide scalable strategies for gathering performance data. A suitable approach must minimize perturbations and reduce the amount of data gathered, while at the same time providing a significant coverage of the important code regions. Studies have shown that the overheads of profiling and tracing[3] are significant when a shared file system becomes a contention point. Tracing can lead to very large trace files and may degrade overall system performance, negatively affecting other applications sharing the same resources.

The use of compilers to support instrumentation has been limited because of portability issues, lack of standard APIs, selective instrumentation and user control. A compiler that has all the necessary language and target support is desired. PDT[4] is a toolkit that was designed in an attempt to overcome this. It gathers static program information via a parser and represents it in a portable format suitable for use in source code instrumentation. This approach does not permit full exploitation of compiler technology; in particular, it does not provide any insight into the compiler translation, or enable the evaluation of any assumptions the compiler makes when deciding to apply a given transformation (which includes metrics used to evaluate static cost models) that may improve the interpretation of performance data. In this paper we show how an open source compiler can be enhanced to enable it to perform selective instrumentation and, as a result, significantly reduce performance measurement overheads.

2 Related Work

Programs may be instrumented at different levels of a program's representation: at the source code level, at multiple stages during the compiler translation, at object code level, and embedded within the run time libraries. All of these techniques have inherent advantages and disadvantages with respect to their ability to perform the instrumentation efficiently and automatically, the type of regions that can be instrumented, the mapping of information to the source code, and the support for selective instrumentation. Among the many performance tools with instrumentation capabilities are KOJAK[2] and TAU[1], which relies on PDT[4] and OPARI[2]to perform source code instrumentation, and Dyninst[5] which performs object code instrumentation. Open SpeedShop[6] and Paradyn support Dyninst and DPCL but instrumenting at the object code level implies a loss of the semantics of the program like losing loop level information. The EP-Cache project [7] uses the NAG compiler to support instrumentation at procedure and loop levels, but is a closed and source system it has not explored ways to improve instrumentation via compiler analysis. Intel's Thread Checker[8] performs instrumentation to detect semantic problems in an OpenMP application but it lacks scalability as it heavily instruments memory references and synchronization points. Other tools like Perfsuite[9], Sun Analyzer and Vtune[10] rely on sampling to provide profile data. Although sampling is a low overhead approach it requires extra system resources, in some cases needing extra threads/processors to support processor monitoring units, and it focuses on low level data gathering.

3 OpenUH

The OpenUH[11] compiler is a branch of the open source Open64 compiler suite for C, C++, and Fortran 95, supporting the IA-64, IA-32e, and Opteron Linux ABI and standards. OpenUH supports OpenMP 2.5 and provides complete support for OpenMP compilation and its runtime library. The major functional parts of the compiler are the front ends, the inter-language interprocedural analyzer (IPA) and the middle-end/back end, which is further subdivided into the loop nest optimizer, auto-parallelizer (with an OpenMP optimization module), global optimizer, and code generator. OpenUH has five levels of a tree-based intermediate representation (IR) called WHIRL to facilitate the implementation

of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Most compiler optimizations are implemented on a specific level of WHIRL, for example interprocedural array region and dependence analysis analysis is implemented in the high level whirl. Our efforts form part of the official source code tree of Open64 (http://www.open64.net), which make results available to the entire community. It is also directly and freely available via the web (http://www.cs.uh.edu/ openuh).

4 Instrumentation

OpenUH provides a complete compile-time instrumentation module covering different compilation phases and different program scopes. Advanced feedback-guided analyses and optimizations are part of the compiler for sequential tuning. We have designed a compiler instrumentation API that can be used to instrument a program. It is language independent to enable it to interact with performance tools such as TAU and KOJAK and support the instrumentation of Fortran, C and C++.

Compile-time instrumentation has several advantages over both source-level and object-level instrumentation. Compiler analysis can be used to detect regions of interest before instrumenting and measuring certain events to support different performance metrics. Also, the instrumentation can be performed at different compilation phases, allowing some optimizations to take place before the instrumentation. These capabilities play a significant role in the reduction of instrumentation points, improve users' ability to deal with program optimizations, and reduce the instrumentation overhead and size of performance trace files.

The instrumentation module in OpenUH can be invoked at six different phases during compilation, which come before and after three major stages in the translation: interprocedural analysis, loop nest optimizations, and SSA/DataFlow optimizations. Figure 1 shows the different places in the compilation process where instrumentation can be performed, along with the corresponding intermediate representation level.

For each phase, the following kinds of user regions can be instrumented: functions, conditional branches, switch statements, loops, callsites, and individual statements. Each user-region type is further divided into subcategories when possible. For instance, a loop may be of type *do loop*, *while loop*. Conditional branches may be of type *if then*, *if then else*, *true branch*, *false branch*, or *select*. MPI operations are instrumented via PMPI so that the compiler does not instrument these callsites. OpenMP constructs are handled via runtime library instrumentation, where it captures the fork and joint events, implicit and explicit barriers[12]. Procedure and control flow instrumentation is essential to relate the MPI and OpenMP-related output to the execution path of the application, or to understand how constructs behave inside these regions.

The compiler instrumentation is performed by first traversing the intermediate representation of an input program to locate different program constructs. The compiler inserts instrumentation calls at the start and exit points of constructs such as procedures, branches and loops. If a region has multiple exit points, they will all be instrumented; for example, goto, stop or return statements may provide alternate exit points. A brief description of the API can be found in Ref.[13].

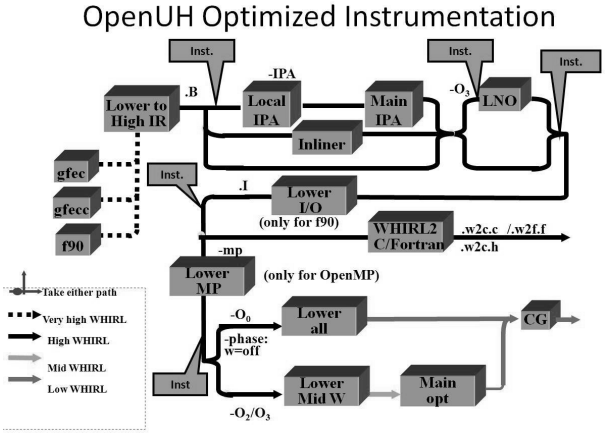

Figure 1. Multiple stages of compiler instrumentation.

5 Selective Instrumentation Analysis

We take advantage of the interprocedural analysis within the compiler to reduce the number of instrumentation points. Here the compiler performs inlining analysis, which attempts to determine where the program will benefit from replacing a procedure call with the actual code of the called procedure's body. As part of this, the compiler must determine if a procedure is invoked frequently and whether the caller and callee meet certain size restrictions in order to avoid code bloat. We adapted this methodology to enable selective instrumentation, for which we have defined a cost model in the form of scores to evaluate the above conditions. We avoid instrumenting any procedure that meets the criteria for inlining. We do instrument procedures that are significant and are infrequently called and have large bodies. We call a procedure significant if it contains many callsites and is well connected in the callgraph.

Our cost model consists of three metrics in the form of instrumentation scores. The first metric computes the *weight* of the procedure using the compiler's control flowgraph, which is defined as $PU_{weight} = (5 * total\ basic\ blocks) + total\ statements + total\ callsites$. As can be seen, this metric puts emphasis on procedures with multiple basic blocks. If runtime information is known, the PU_{weight} formula will use the number of times or *effective* number of basic blocks, statements and callsites invoked at runtime. The other metric we use is the frequency with which a procedure is invoked, taking their position within loop nests into account. The formula used is: $PU_{loop-score} = (100 - loopnest\ level) * 2048$. This formula gives higher scores to procedures invoked with fewer nesting levels. The third metric is a score that quantifies how many calls exist within a procedure. $PU_{callsite-score} = (callsites\ in\ callee) * 2048^2$. This formula gives a small score to procedures invoked as leaf nodes in the callgraph or that have few calling edges. The constants of the formulas were determined empirically based on the inlining algorithm of the compiler which was tunned to avoid under or over inlining. Our assumption here is that important procedures are connected with others, and thus are associated with several edges in the callgraph. It is important to note that we will not count callsites to procedures

that are not going to be instrumented. The overall score used to decide whether we will instrument a procedure is as follows:

$$Instrumentation\ Score = PU_{weight} + PU_{loop-score} + PU_{callsite-score}$$

Our strategy for computing this score means that we will favour procedures with large bodies, invoked few times and with multiple edges connecting them to other procedures in the callgraph. We avoid the instrumentation of small procedures invoked at high loopnest levels and that are leaf nodes in the callgraph.

With this score we then define a threshold that can be changed depending on the size of the application, in order to avoid over or under-instrumentation. Also, we generalize our approach to take into consideration the lowest score that a procedure has from its different callsites. If a score for a procedure is below a pre-defined threshold, the procedure will not be instrumented.

$$Instrument\ Procedure < Threshold < Do\ not\ Instrument$$

Table 1 contains the instrumentation scores for some of the procedures in the BT OpenMP benchmark from the NAS parallel benchmarks. It shows that procedures corresponding to leaf nodes in the callgraph have a low instrumentation score. Heavily connected nodes in the callgraph (those that have several callsites) are among the ones with the highest score, as is to be expected. If we define a threshold to be 204800 (the score of an empty procedure with no callsites being invoked outside a loop), then we will not instrument the following procedures: *matvecsub, binvchrs, matmul_sub, lhsinit, exact_solution* and do instrument the following procedures: *adi, x_solve, y_solve, z_solve, main*.

Table 1. Instrumentation scores for the BT OpenMP benchmark

Proc.	Weight	Loop Score (level)	Callsite Score (sites)	Inst. Score
matvecsub	23	198656(3)	0(0)	198679
binvchrs	240	198656(3)	0(0)	198896
binvrhs	115	198656(3)	0(0)	198771
matmul_sub	27	198656(3)	0(0)	198683
lhsinit	57	198656(3)	0(0)	198713
exact_solution	23	196608(4)	0(0)	196631
adi	45	202752(1)	20971520(5)	21174317
x_solve	278	204800(0)	41943040(10)	42148118
y_solve	278	204800(0)	41943040(10)	42148118
z_solve	278	204800(0)	41943040(10)	42148118
main	459	204800(0)	58720256(14)	58720256

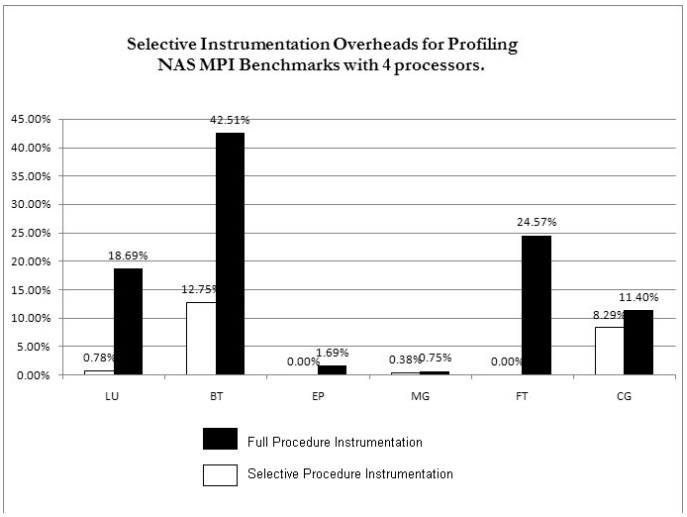

Figure 2. Selective instrumentation in the NAS MPI Benchmarks.

6 Experiments

We applied the the selective instrumentation algorithm to six benchmarks from the NAS parallel benchmarks in both MPI and OpenMP implementations. Our experiments used the class A problem size and were conducted on an SGI Altix 4000 with 16 Itanium 2, 1.6 Ghz processors, and the NUMAlink interconnect. Figures 2 and 3 shows the overheads incurred when performing full procedure instrumentation versus performing selective instrumentation when using the TAU profiling libraries. For the MPI benchmarks we turned off TAU throttle. The purpose of THROTTLE is to disable the instrumentation library at runtime when a procedure reaches a given threshold.

Because of the massive overheads of full instrumentation in the case of OpenMP, we turned on the TAU THROTTLE for NUMCALLS=300 and PERCALL=300000 environment variables. Selective instrumentation reduces the overheads by an average of 90 times in the OpenMP version even when TAU THROTTLE is enabled in full instrumentation. The FT-OMP benchmark has particularly high overheads compared to the other benchmarks.

This is because it invokes the procedures *fttz2, cfftz* a significant number of times. We note that *cfftz* calls *fttz2*. Our selective instrumentation score for *cfftz* is 202817 and *fftz2* is 198772 , which is below our instrumentation threshold. As a result, we also do not instrument the *cfftz* procedure since it only has one callsite that is not being instrumented. In the CG benchmark we do not instrument the procedures *buts, jaci, blts, jacld and exact* which are leaf nodes in the callgraph that are invoked many times and have small weights. The higher overheads in OpenMP instrumentation versus MPI are due to memory contentions and locks on where the performance data is stored and modified. In the Altix this contention becomes a problem due to cache line invalidations and remote memory accesses due to the cc-NUMA architectures.

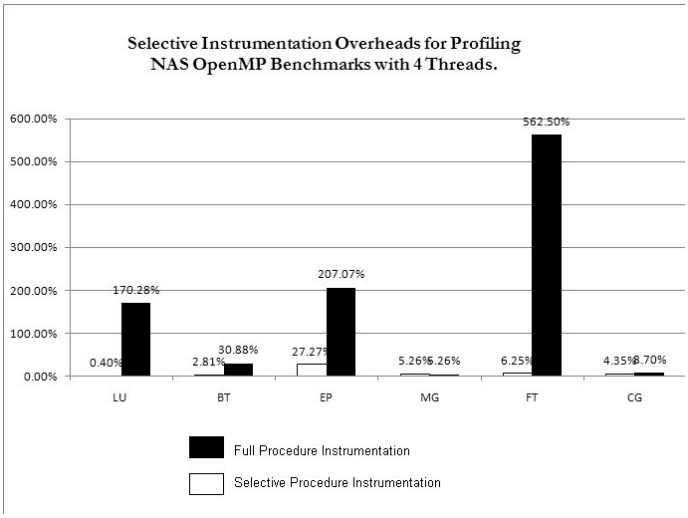

Figure 3. Selective instrumentation in the NAS OpenMP Benchmarks.

We also applied our algorithm to an MPI implementation of a Cloud-Resolving Model code[14] using a grid size of 104x104x42 and 4 MPI processes for our experiment. Our selective algorithm determined that we should not instrument three leaf procedures in the callgraph. We were able to reduce the profiling overhead from 51% to 3% based on this alone.

7 Conclusions and Future Work

In this paper we have presented a selective instrumentation algorithm that can be implemented in a compiler by adapting a typical strategy for performing inlining analysis. In the examples presented here, selective instrumentation based upon this algorithm was able to reduce profiling overheads by 90 times on average in the NAS OpenMP parallel benchmarks and 17 times for the cloud formation code. Our future work will combine feedback-directed optimizations and the inlining analysis to further improve selective instrumentation. We will also continue to explore opportunities to enhance the working of performance tools via direct compiler support.

8 Acknowledgements

We would like to thank NSF for supporting this work[a]. We will also want to thank Bob Hood and Davin Chang from CSC at Nasa Ames for providing us the Altix System to perform the experiments.

[a]This work is supported by the National Science Foundation, under contract CCF-0444468

References

1. Allen D. Malony, Sameer Shende, Robert Bell, Kai Li, Li Li and Nick Trebon, *Advances in the TAU performance system*, Performance analysis and grid computing, pp. 129–144, (2004).
2. B. Mohr and F Wolf, *KOJAK - a tool set for automatic performance analysis of parallel applications*, in: Proc. European Conference on Parallel Computing (EuroPar), pp. 1301–1304, (2003).
3. K. Mohror and K. L. Karavanic, *A study of tracing overhead on a high-performance linux cluster*, in: PPoPP '07: Proc. 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 158–159, (ACM Press, New York, 2007).
4. K. A. Lindlan, J. E. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh and C. Rasmussen, *A tool framework for static and dynamic analysis of object-oriented software with templates*, in: Supercomputing, (2000).
5. B. Buck and J. K. Hollingsworth, *An API for runtime code patching*, in: International Journal of High Performance Computing Applications, **14**, 317–329, (2000).
6. M. Schulz, J. Galarowicz and W. Hachfeld, *Open—SpeedShop: open source performance analysis for Linux clusters*, in: SC '06: Proc. 2006 ACM/IEEE Conference on Supercomputing, p. 14, (ACM Press, NY, 2006).
7. E. Kereku and M. Gerndt, *Selective instrumentation and monitoring,*, in: International Workshop on Compilers for Parallel Computers, CPC'04, (2004).
8. P. Petersen and S. Shah, *OpenMP support in the Intel thread checker*, in: WOMPAT, pp. 1–12, (2003).
9. R. Kufrin, *PerfSuite: an accessible, open source, performance analysis environment for Linux*, in: 6th International Conference on Linux Clusters (LCI-2005), Chapel Hill, NC, (2005).
10. J. H. Wolf, *Programming methods for the Pentium III processor's streaming SIMD extensions using the VTune performance enhancement environment*, Intel Technology Journal, no. Q2, 11, (1999).
11. C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng, *OpenUH: an optimizing, portable OpenMP compiler*, in: 12th Workshop on Compilers for Parallel Computers, (2006).
12. V. Bui, O. Hernandez, B. Chapman, R. Kufrin, D. Tafti and P. Gopalkrishnan, *Towards an implementation of the OpenMP collector API*, in: PARCO, (2007).
13. O. Hernandez, F. Song, B. Chapman, J. Dongarra, B. Mohr, S. Moore and F. Wolf, *Instrumentation and compiler optimizations for MPI/OpenMP applications*, in: International Workshop on OpenMP (IWOMP 2006), (2006).
14. H.-M. H. Juang, W.-K. Tao, X. Zeng, C.-L. Shie, S. Lang and J. Simpson, *Implementation of a message passing interface into a cloud-resolving model for massively parallel computing*, in: Monthly Weather Review., (2004).